Chapter 10

結論

前章までで、Nutsの構造／制御モデルに基づいたコンポーネントアーキテクチャを用いて、実用的なアプリケーションの構築が可能であることを示した。また、Nutsを用いて開発したアプリケーションは、ホワイトボックス的な拡張とコンポーネント差分プログラミングによって、高い柔軟性を保持するとともに、開発したコンポーネントを容易に再利用対象とできることを述べた。本章では、Nutsの柔軟性の向上や、Nutsの普及を考慮する上での、今後の課題についてまとめ、結論を述べる。

10.1 Nutsフレームワークの今後の課題

10.1.1 ベクターを修飾するベクター

現在、ベクターコンポーネント自体の拡張は、サブクラス化によって行う。これに対して、ベクターコンポーネントと結合し、ベクターコンポーネントの機能を拡張するメタベクターコンポーネントも可能性としては考えられる。そもそもベクターコンポーネントの利点として、ベクターコンポーネントが被拡張コンポーネントのクラス階層と独立したクラス階層を持っていることにより、拡張機能の開発が容易なことが挙げられる。しかし、ベクターコンポーネントの種類が増え、そのクラス階層が複雑になってくると、ベクターコンポーネントのクラス階層を理解するのが困難になる。そうした場合に、ベクターコンポーネントのクラス階層を理解せずに、ベクターコンポーネントを拡張できるコンポーネントがあればより扱いやすい。つまり、ベクターコンポーネント自体の拡張をコンポーネント差分プログラミングで行う。

しかし、この方法では、上記のベクターベクターコンポーネントをさらにコンポーネント差分プログラミングで拡張する必要が生じるかもしれない。この問題は、ベクターコンポーネントのクラス階層を管理／理解するコストと、多段のコンポーネント差分プログラミングを用いた拡張のためのプログラミングコストのトレードオフに帰着すると考えられる。今後、この点を踏まえながら実現可能性を検討する。

10.1.2 コンポーネント内部の拡張

現在のコンポーネント差分プログラミングは、メッセージの横取りをベースとした方法で行っている。メッセージによる制御モデルに従えば、第9.2.1節で示したような抽象メソッドの拡張もコンポーネント差分プログラミングによって行える。しかし、現在のC++をベースとしたNutsフレームワークでは、メッセージに変換されないコンポーネント内部での直接的なメソッド呼び出しも容認されており、これらはコンポーネント差分プログラミングの対象とならない。

これに対して、あらゆるメソッド呼び出しをNutsのメッセージに置き換える仕組みがあれば、コンポーネントの内部仕様の拡張も可能になる。例えば、C++のマクロ的な方法で、一般の関数呼び出しが、Nutsの横取り機構で横取り可能な関数に置き換えられような仕組みが必要となる。

これに適した手法として、第9.4節で説明した、コンパイル時MOPが役立つと考えられる。例えば、コンパイル時MOPの一つであるOpenC++ [11] を用いれば、C++のメソッド呼び出しである "->" オペレータをNutsのメッセージ呼び出である "`sendMessage(...)`" に置き換えられる。

例として、クラスPersonに対する、C++の "."(ドット)オペレータによるメソッド呼び出しを、"`receiveMessage(...)`" で横取りするには、図10.1のようにPersonClassメタクラスを定義する。これにより、図10.2を実行すると、クラスPersonの`Age()`や`BirthdayComes()`が直接呼び出されるのはなく、`receiveMessage()`が呼び出され、以下のような結果が得られる。

```
Age=24
Age=25
```

すなわち、メタクラスを定義することにより、通常のドットオペレータによるメソッド呼び出しを、Nutsのメッセージ呼び出し的な方法に置き換えられる。今後は、Nutsの制御モデルをより広範囲に適用させる方法として、コンパイル時MOPのメタクラスの利用を検討する。

10.1.3 適用範囲の拡大

第2.8節で説明したように、現在のNutsは、C++の開発フレームワークとしての性格が強いが、コンポーネントアーキテクチャ自体は、その他のオブジェクト指向言語にも対応できる。例えば、現在のオブジェクト指向言語の主流となっているJavaに対応したNutsコンポーネントを開発することもできる。

実験的にNutsの基底クラスとJavaのAWT部品のいくつかをNutsコンポーネントとして実装した例を表10.1に示す。また、これらのJava版Nutsコンポーネントを用いたコーディング例とその実行結果を図10.3に示す。Java上でもNutsの構造モデルにしたがったプログラムの構築が可能なことがわかる。

さらに、第8章で示したNavimosアプリケーションのように、現在の中心的な実行環境である、Win32APIへの対応も重要である。MFC[66] のコントロール群をNutsコンポー

```
#include "mop.h"
class PersonClass : public Class {
public:
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*,
                               Ptree*, Ptree*);
};
Ptree* PersonClass::TranslateMemberCall(Environment* env,
        Ptree* object, Ptree* op, Ptree* member, Ptree* arglist)
{
  return Ptree::Make("%p.receiveMessage(\"%p\")",object,member,
       Class::TranslateMemberCall(env, object, op,
  member, arglist));
}
```

図 10.1: OpenC++のメタクラスの定義

```
#include <stdio.h>

metaclass PersonClass Person;
class Person {
private:
    int age;
public:
    Person(int i) { age = i; }
    int Age() { return age; }
    int BirthdayComes() { return ++age; }
    void receiveMessage(char *);
};
void Person::receiveMessage(char *message)
{
  if(strcmp(message,"Age") == 0){
    fprintf(stdout,"Age = %d\n",age);
  }
  else if(strcmp(message,"BirthdayComes") == 0){
    fprintf(stdout,"Age = %d\n",++age);
  }
}
void main()
{
    Person billy(24);
    billy.Age();
    billy.BirthdayComes();
}
```

図 10.2: OpenC++のメタクラスの使用

```
public class nuts extends NutsMain{
  static final NutsCore  app     = new NutsApp("nuts");
  static final NutsCore  frame   = new NutsFrame(app,"nuts-frame");
  static final NutsCore  mb      = new NutsMenuBar(frame,"menu_bar");
  static final NutsCore  file    = new NutsMenu(mb,"File");
  static final NutsCore  open    = new NutsMenuItem(file,"Open");
  static final NutsCore  save    = new NutsMenuItem(file,"Save");
  static final NutsCore  saveas  = new NutsMenuItem(file,"Save As");
  static final NutsCore  sp1     = new NutsMenuItem(file,"-");
  static final NutsCore  quit    = new NutsMenuItem(file,"Quit");
  static final NutsCore  edit    = new NutsMenu(mb,"Edit");
  static final NutsCore  cut     = new NutsMenuItem(edit,"Cut");
  static final NutsCore  copy    = new NutsMenuItem(edit,"Copy");
  static final NutsCore  paste   = new NutsMenuItem(edit,"Paste");
  static final NutsCore  help    = new NutsMenu(mb,"Help");
  static final NutsCore  about   = new NutsMenuItem(help,"About this");
  static final NutsCore  nuts    = new NutsButton(frame,"nuts");
  static final NutsCore  tx      = new NutsTextField(frame,"nutsj text");
};
```

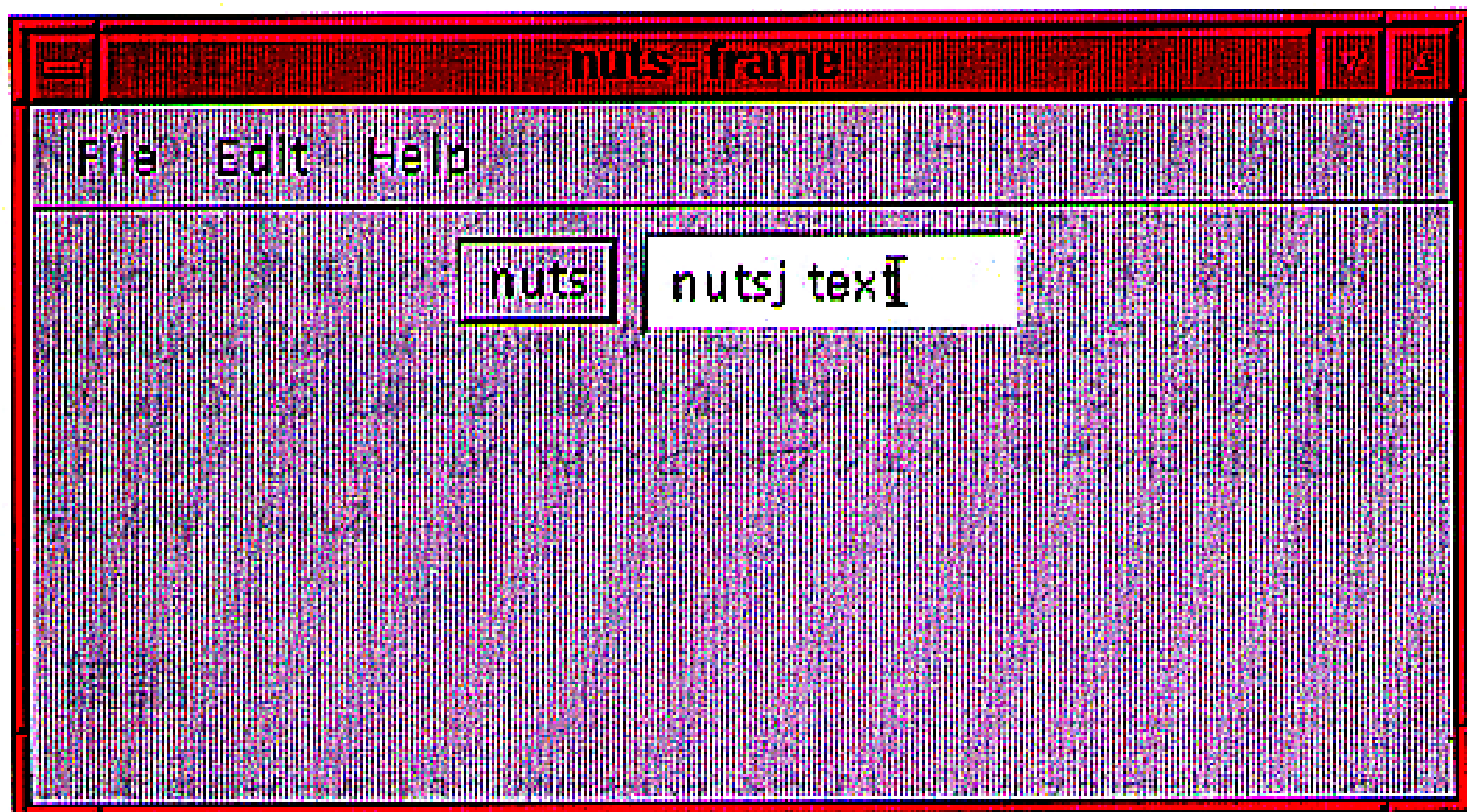

図 10.3: Java 版 Nuts のコーディング例と実行結果

表 10.1: Java 版 Nuts のコンポーネント群

コンポーネント名	コード行数
NutsApp.java	131
NutsButton.java	235
NutsComponent.java	448
NutsContainer.java	331
NutsCore.java	607
NutsFrame.java	808
NutsMain.java	139
NutsManager.java	502
NutsMenu.java	397
NutsMenuBar.java	404
NutsMenuItem.java	247
NutsPanel.java	223
NutsTextField.java	252

ネントとしてカプセル化すれば、Win32 上でのアプリケーション開発がより迅速に行えると考える。

10.1.4 分散化

現在の Nuts の実装では、同じ計算機上にあるコンポーネント同士でなければ結合できない。これに対して、ネットワーク環境下で分散処理を行うには、DCOM[37] や CORBA[34] のように異なった計算機上のコンポーネントでも互いに結合できる必要がある。そのためには、ORB(Object Request Broker) などによって探索可能なコンポーネントの仕様と、Skeleton、Proxy など分散環境を意識させない代理コンポーネントが必要になる。

現実的には、先行している DCOM などのオブジェクトモデルに自動変換するツールを開発する方法が考えられる。

10.2 結論

オブジェクト指向プログラミングにおいてホワイトボックス的であり、かつブラックボックス的なプログラミングが可能な形態としてホワイトボックス的なコンポーネントを提案した。そして、ホワイトボックス的なプログラミングが可能なコンポーネントの実施例である Nuts コンポーネントアーキテクチャについて説明した。

Nuts コンポーネントでは、すべての部品を同じ方法で扱えるように構造と制御をモデル化した。このモデルによって、従来のデザインパターンやフレームワークの利用ではまち

まちであった、コンポーネント結合とそれらの間の通信を一貫性のあるものにした。その上で、Nuts コンポーネントは、木構造モデルとメッセージの動的探索による通信を用いることで、拡張性や柔軟性を維持している。また、それぞれのコンポーネントは、ホワイトボックスであり、サブクラス拡張によって新たなコンポーネントを作ることができる。

さらに、Nuts におけるコンポーネントベースのプログラミングを進化させ、コンポーネントレベルでの差分プログラミングが可能なようにアーキテクチャを拡張した。Nuts で提案するコンポーネント差分プログラミングでは、ベクターコンポーネントと呼ばれる拡張機能をカプセル化したコンポーネントを開発し、プログラム中のコンポーネントと結合させることで拡張機能を付加できる。コンポーネント差分プログラミングでは、被拡張コンポーネントとベクターコンポーネントが別のクラス階層に属しており、被拡張コンポーネントのクラス階層を熟知していなくても拡張機能の開発が可能である。また、サブクラス拡張の多重継承によらずに、多重の機能拡張が可能である。

実際のソフトウェアの開発事例を通して、Nuts の効果を述べた。コード量、工数、再利用の対象となる部品の割合などの観点から、Nuts を用いたソフトウェア開発が、効率的であることを示した。

また、Nuts のアーキテクチャは、デザインパターンの Decorator、Strategy、Mediator、Facade などのパターンを一般化し、デザインパターンには示されない、実装上の問題においても有効な手法を提供していることを示した。

今後は、部品の拡充、分散化、他プラットフォームへの移植等を通して、実際のソフトウェア開発に役立つフレームワークを構築し、アプリケーション開発の適用事例を増やしていきたい。